An Event-driven Multi-threading Real-time Operating System dedicated to Wireless Sensor Networks

Hai-ying Zhou, Feng Wu
School of Computer Science & Technology, Harbin Institute of Technology, Harbin, China
haiyingzhou@hit.edu.cn , wufeng@ftcl.hit.edu.cn

Kun-mean Hou
LIMOS Laboratory UMR 6158 CNRS, University of Blaise Pascal, Clermont-Ferrand, France
kun-mean.hou@isima.fr

Abstract

At present, the OSs (Operating system) employed for WSN (wireless sensor networks) are either satisfied with only one or two application classes or unsuitable for strict-constrained resources. In view of a variety of WSN applications, there is a need of developing a self-adaptable and self-configurable embedded real-time operating system (RTOS). This paper presents a resource-aware and low-power RTOS termed LIMOS. This kernel adopts a component-based three-level system architecture: action (system operation), thread (component) and event (container). In accordance, a predictable and deterministic two-level scheduling mechanism is proposed: 'non pre-emption priority based' high level scheduling for events and 'preemptive priority-based' low level scheduling for threads. Employing the concepts of LINDA language, LIMOS provides a simplified tuple space and a light IN/OUT system primitive-pair to achieve system communication and synchronization. LIMOS is capable of self-adapting to run on two operation modes: event-driven and multi-threading, with respect to the application diversity. The performance evaluation and comparison shows LIMOS has tiny resource consumption and is fit for the real-time applications. Currently LIMOS has been ported on several hardware platforms for different WSN applications.

1. Introduction

The emergence and development of WSN (Wireless Sensor Networks) technology explore new issues and challenges in a variety of traditional fields such as wireless communication, sensors and embedded system, etc. How to design a dedicated EOS (embedded operation system) for a WSN system is one of the main challenges.

The huge potential of WSN applications needs the EOS to be suitable for different operating environments, from a simple single-task event to a real-time multi-thread system. Moreover, due to the resource constrains of WSN node, the EOS must consume tiny resource, including CPU, and memory. It means that the EOS must be resource- and context-aware to minimize energy consumption.

Traditional EOSs, such as SDREAM[2], µC/OS-II[4], VxWorks[4],QNX[5][9], pSOS[6], Lynxos[7], RTLinux[8], WinCE.NET[10], RTX[11], HyperKernel[12] etc, are generally not fit for WSN as they cannot satisfy both of the above-mentioned requirements. For example, TinyOS0, the known WSN dedicated OS which adopts event-driven component-based structure, has tiny resource consumption. However, it is an essential single task system and not fit for multi-threading application. On the other hand, other multi-threading scheduling OSs are either unsuitable for a variety of environments (partially efficient) or consume overfull resources. For example, SDREAM[2], a typical multi-threading OS, is suit for real-time multi-task scheduling. But in view of the multi-event single task applications, adopting multi-threading scheduling mode has more resource consumption and less efficiency comparing to the event-driven mode. Our objective is to design a general OS dedicated to various WSN applications.

LIMOS (LIghtweight Multi-threading Operating System) is a native configurable hybrid operating system that can thus operate in either event-driven mode or multi-threading mode to minimize resource requirement and improve system efficiency according to practical application environments. The following sections introduce LIMOS in details: section 2 describes system architecture; section 3 depicts a two-level system scheduling policy and section 4 presents the system communication and synchronization mechanism; section 5 presents the performance

978-0-7695-3287-5/08 $25.00 © 2008 IEEE
DOI 10.1109/ICESS.2008.58

evaluation and the last section concludes LIMOS and gives perspectives.

2. System architecture

Combining the concepts of event-driven and multi-threading systems, LIMOS adopts a component-based multi-level system architecture: action, thread and event. In LIMOS, the minimal system unit is termed action that responds to the basic system operation, including 'read', 'write', 'schedule', etc. A thread is a component that consists of a set of actions, which represents a specific task. An event, which is a super-component (container) having a set of components , is an independent job that consists of a set of tasks (threads).

2.1. Component-based multi-level structure

2.1.1. Action. Action is the minimal system unit that is classified into two classes: system action (device-dependent) and function action (device-independent). To illuminate by serving as an example of WSN, there are generally four types of actions: 'read' operation (reading data from devices), 'analyse' operation (processing these acquisitions), 'write' operation (transmitting the results) and 'scheduler' operation (thread/event scheduler).

The basic actions have been implemented and stored into the action library. Threads call an action directly like calling a library function. Since actions shield the differences of hardware devices and provide a uniform calling interface for components, system designers can thus focus on functions development with no need of concerning about device-dependent operations. This feature simplifies the system design and also improves the system reusability and compatibility.

2.1.2. Thread. Thread represents a task, and interacts only with the components within the same container (event). Threads share resources within the containers and communicate with one another via shared space. Threads run in concurrence or in parallel to realize a job (event) by interacting with each other. Hence, threads are pre-emptive and a thread thus needs a stack space to store its private "contexts" information. Sharing resources within the same event reduces the overheads of thread switching and also decreases the costs of system resources.

2.1.3. Event. Event represents an independent job that interacts with peripheral devices or other events. Events are signal-driven. An event is activated only after receiving a trigger signal from an ISR (Interrupt Service Routine). LIMOS defines that each event can be associated with one and only one input interrupt source and n output input sources ($0<n\leq Max$). LIMOS events run to completion without preemption. Therefore events are non-preemptive. In terms of the regularity of occurrences, events are divided into two classes:

- *Periodic* events: typically suitable for regular data sampling issued from sensors or for monitoring actuators, which occur at a fixed rate. Periodic events have certain execution time and deterministic response time.
- *Sporadic* events: typically suitable for strict time-constrained tasks, which occur sporadically. Sporadic events can be used to deal with the critical problems.

2.1.4. Architecture. A set of actions construct a component (thread) and a set of components make up of a container (event). An event can be viewed as an automaton. Once being triggered by a signal, the event is activated and its threads start to be executed. Events receive/send a signal by calling *IN/OUT* system primitives via their related *tuple* spaces. Providing an inside view, an event is a set of threads which operate concurrently and cooperatively. Threads adopt the same mechanism for inter-thread communication and synchronization.

Denoting $\Re$ is a LIMOS instance, ε is an event, τ is a thread, α is an action and ς is a signal. The '$\longrightarrow$' symbol denotes precedence sequential operation and '$\|$' represents the concurrent or parallel operation. The LIMOS architecture can be expressed as follows:

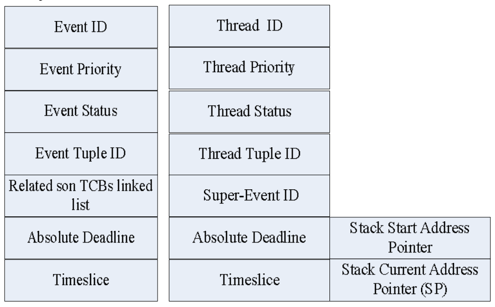
$$\begin{aligned} \Re &= \{\varepsilon_i : i = 1,2,\cdots,n;\ \varepsilon_1 \rightarrow \varepsilon_2 \cdots \rightarrow \varepsilon_n\}; \\ \varepsilon_i &= \{\tau_{ij} : j = 1,2,\cdots m_i;\ \tau_{i1} \| \tau_{i2} \cdots \| \tau_{im_i}\}; \\ \tau_{ij} &= \{\alpha_{ijk} : k = 1,2,\cdots S_{ij};\ \alpha_{ij1} \rightarrow \alpha_{ij2} \cdots \rightarrow \alpha_{ijS_{ij}}\} \end{aligned} \quad (1)$$

LIMOS can be configured to run independently in event driven, multi-thread and event driven with multithreading (hybrid) modes. Considering two extreme-case scenarios: if $\Re$ has only one event, i.e. n=1, LIMOS works in the multi-threading mode as SDREAM; while if each event has just one thread, i.e. $\forall m_i=1,\ i\in\{1,\cdots,j\}$, LIMOS works in the event-driven mode as TinyOS.

2.2. Event-/thread-control Block

In order to simplify the management and scheduling, LIMOS allocates two data table structures, named event control block (ECB) and thread control block (TCB), to store events and threads respectively. Each data structure contains at least an identification flag (string or number), a priority, a status, and a corresponding *tuple* ID. Each structure has a specific attribute to depict its affiliation relation: the sub-TCBs linked list pointer of ECB and the super-Event ID of TCB. ECBs and TCBs also contain two time parameters to describe the absolute deadline and the allowable time-slice. For events, when an event is defined and activated, the initial value of time-slice, which is decreasing with the time passing, is equal to the maximum allowable response time.. These two parameters are used to evaluate the event priority and then to determine event scheduling. For threads, the two parameters are used to avoid the thread deadlock. TCBs contain two more attributes, i.e. start stack pointer (SSP) and current stack pointer (CSP), to indicate a thread's stack resource space. The stack space is used to store program counter (PC), status register(s), general registers and various variables.

The essential information of ECB and TCB are shown in Figure 1. To simplify its implementation, LIMOS stores the ECBs and TCBs in two fixed-size arrays rather than linked lists since the numbers of real-time tasks (events, threads) in WSN applications are generally known and limited.



Event ID	Thread ID	
Event Priority	Thread Priority	
Event Status	Thread Status	
Event Tuple ID	Thread Tuple ID	
Related son TCBs linked list	Super-Event ID	
Absolute Deadline	Absolute Deadline	Stack Start Address Pointer
Timeslice	Timeslice	Stack Current Address Pointer (SP)

Figure 1. Event-control block (left) and Thread-control block (Right)

2.3. Event/thread states

LIMOS manages ECBs / TCBs by keeping track of the states of each event / thread. System scheduling can be achieved by manipulating the states status. In LIMOS, events and threads have similar states except that threads have one more state: 'suspended'.

- *Sleep*: An event or thread was created and initialized, but it is not yet ready and eligible to execute. Since LIMOS is configured statically and the numbers of events/threads are fixed, an event changes state from 'sleep' to 'terminated', where sub-threads are reinitialized correspondingly.
- *Ready*: Events or threads belong to this state are those that are released and eligible for execution, but are not executing. An event or thread enters the 'ready' state from 'sleep' state when its trigger signal occurs. A thread can also enter 'ready' from 'suspend' if its waiting resource is available.
- *Execute*: When an event/thread is being executed. Due to its non-preemptive feature, an event runs to completion with no change of its state.
- *Suspend*: Only threads can enter this state. If a thread which is executing waits for an unavailable resource, or it was preempted, then the thread is suspended and 'suspend' state is assigned.
- *Terminate*: A thread has self-terminated or is aborted. If all of its sub-threads are terminated, the event enters the 'terminated' state.

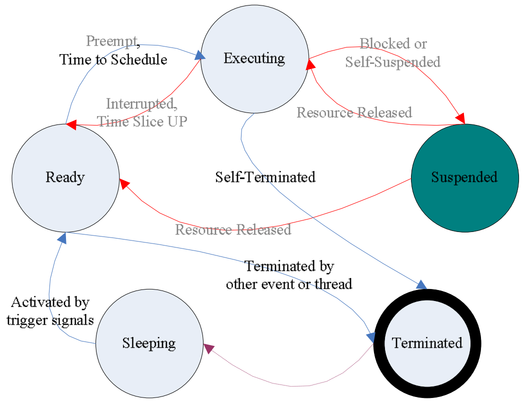


Figure 2. An event and thread state diagram

A state diagram corresponding to event and thread states is depicted in Figure 2. The state transition operations exclusively related to threads are traced in red lines and explained by gray texts. The unique broken line represents the state transition from 'terminate' to 'sleep'. It should be noted that, according to the features of the static pre-configuration and the determinate numbers of events/threads, LIMOS can combine the predefined 'terminate' and 'sleep' states into the 'idle' state. In this way, an event or thread enters the 'idle' state after it has finished execution, but the event resource is not released and the event will be activated by next signal.

3. System Scheduling

Scheduling is a fundamental operation of an operating system. In order to meet a program's temporal requirements of a real-time system, it is of the utmost importance that the scheduling algorithm should produce a predictable schedule, that is, at all times it should be known that which task is going to be executed. With event/thread system architecture, LIMOS offers a two-level dynamic-priority scheduling scheme: 'non-preemptive-priority' for events at high level and 'preemptive-priority' for threads at low level.

3.1. Event-/thread-control Block

3.1.1. Event scheduling. Event scheduling adopts the priority-based non-preemptive scheduling and I/O devices are interrupt-driven. An event is elected to run according to its priority and an active event runs to completion without any pre-emption. At any instant when finish an event, the event scheduling is activated to elect the next event to be executed. This scheme is a dynamic-priority scheduling, like earliest-deadline-first (EDF) algorithm [13]: the event with earliest deadline has the highest priority.

Note that events are interrupt-able. When an interrupt occurs, its ISR stores the corresponding message (*msg*) or signal and then sends it to the corresponding *tuple* buffer by calling the *OUT* primitive. The further processing is reacted only after the active event has run to completion. Meanwhile, the *OUT* primitive updates the attribute values of the event, including the *status*, the *time-slice*, and also revaluates the priorities of all ready events according to the scheduling scheme.

For a periodic event, the relative deadline, D_i, is supposed to equal its period p_i; while for a sporadic event, the system designates a maximum allowable response time, D_i, when the event occurs. A sporadic event generally can be regarded as the first instance of a periodic event with the period of D_i. Therefore, $d_{i,k}$, the absolute deadline of the *k*th instance of the event ε_i is as follows:

$$d_{i,k} = r_{i,k} + p_i = \phi_i + k * p_i \qquad (2).$$

Hence, each time a signal is sent to the corresponding *tuple* space, the *status* of the event changes from 'idle' to 'ready', the *absolute deadline* is $d_{i,k}$ and the *time-slice* is p_i. The *time-slice* value decreasing with the time passing is used to evaluate the event priority: the smaller the *time-slice* value is, the higher the event priority is.

3.1.2. Thread scheduling. Thread scheduling adopts a priority-based pre-emptive scheduling scheme. Threads are elected to run in the order of priority and the elected thread can pre-empt any other lower priority threads at any execution point outside of system critical section. When the threads of the active event are running, the threads of other events are not eligible to compete to obtain CPU resource. Thus it allows events to run until completion.

The thread scheme is a static priority scheduling and the thread priority is predefined at pre-run-time and is fixed according to the relationships of threads. The priorities of threads must be assigned carefully to avoid deadlock situation. In the case of deadlock, two or more threads cannot proceed due to circular wait. For example, supposing that a high priority thread (τ_h) is held on its corresponding tuple, while only a lower priority thread (τ_l) is allowed to send data (message or signal) to wake up τ_h. If at the same time, τ_l is held on its tuple which only τ_h will send data to wake up. In this condition, the two threads enter a dead-circular situation. Hence, in order to produce a predictable scheduling and to allow a determinative execution time, the priorities of threads are assigned as below:

1. Considering a thread which is held-to-wait a signal of an event ε_i, is the successor thread τ_{i0} of ε_i. Then, τ_{i0}, has the highest priority;
2. Supposing each thread will be held on one and only one thread *tuple*.
3. Therefore, the thread $\tau_{i,j}$ that is waiting on the *tuple* message (*msg*) from its previous thread $\tau_{i,j-1}$, has the lower priority than that of $\tau_{i,j-1}$, and so on. For example, the thread that is waiting an *msg* from the successor thread has the second high priority.

Threads are interruptable and preemptive. A thread interacts with other threads via tuple by calling the *IN/OUT* primitive-pair. When a thread sends a message or signal to a tuple, the *OUT* primitive is called to change the state of the tuple-relative thread from 'idle' or 'suspend' to 'ready'. Moreover, the time-slice value of each item of TCB suspend list is recalculated in view of the current timeslot and the deadline value.

3.1.3. Event-to-thread jump. The following theorem gives the condition under which a feasible schedule exists under the EDF scheme .

Theorem [EDF Bound][13]: A set of *n* periodic events, each of whose relative deadline equals its period, can be feasibly scheduled by EDF if and only if

$$\sum_{i=1}^{n} (e_i / p_i) \le 1 \qquad (3)$$

If a LIMOS instance consists of periodic events and meets the formula(3), its event/thread scheduling scheme can be predictable. If the execution time of events is deterministic and transient, LIMOS can be considered as a soft real time system. But for hard real-time events, general being sporadic events, which need to be reacted within a strict time-constrained deadline, the above-defined scheduling scheme does not always work well if the formula (4) cannot be satisfied.

Let ε_s and ε_p be the sporadic event and the active periodic event, D_s and e_s be the relative deadline and the execution time of ε_s, and e_p be the expected time from the interrupted instant to the completed instant of

ε_p, where $0 < e_p \le e_i$. Then the feasibly of the sporadic event ε_s meets its deadline is that

$$D_s < e_p + e_s \qquad (4)$$

Note that since the execution time of ISR routine related to the sporadic event ε_s is very short and deterministic, it is thus ignored. If the formula (4) is not met, therefore, the event ε_s miss its deadline and the system then enters an unexpected exceptional status. To ensure in this worst-case scenario the sporadic event can be handled with short response time to meet its deadline, a special *event-to-thread jump* mechanism is proposed: the sporadic event ε_s is treated as the highest priority thread τ_s of current periodic event ε_p, so that τ_s can preempt any other active thread of ε_p according to the thread scheduling scheme. The *event-to-thread jump* mechanism breaks down the obstacle between threads and events, allowing the threads of an urgent event to preempt the CPU resource.

Let *P(ε)* and *P(τ)* be the priorities of events and threads, then when an *event-to-thread jump* occurs, the priorities of the threads of the events (ε_s and ε_p) is:

$$P(\varepsilon) = P(\varepsilon) + P(\tau)/10 \qquad (5)$$

Note that LIMOS quantifies the priority of events and threads between 0 and 10. Since the sporadic event has higher priority than the active event, that is, $P\varepsilon_s > P\varepsilon_p$, then the threads of ε_s have higher priority than those of ε_p so that they can preempt the CPU resources.

3.2. Scheduling program

The LIMOS two-level scheduling program is implemented by using C language and it has two main parts: the *scheduler* function, which implements event/thread dispatch, will be called regularly at a fixed-rate by Timer ISR or when a thread is terminated; the *IN/OUT* system primitive-pair ,which realizes the priority computation (event only) and ECB/TCB list sorting according to priority and event-to-thread operation (optional), will be executed when calling *IN/OUT* primitives. Two global variables are defined, *cur_event* and *cur_thread*, to indicate current active event and thread.

The pseudo-code programs show that this scheduling scheme is smart and predictable, and has deterministic execution time independent of the event and thread numbers. Note that the top item of ECB ready lists or TCB lists is the one with the highest priority, which has been elected and sorted by the rules of the scheduling scheme in the *IN/OUT* system primitive-pair.

```
void event-to-thread-jump (void)
{
      /* Enable the possibility of event preemptive, being
         suitable for hard real-time events*/
      Insert TCBs of the hard real-time event at the top of
      the TCB list of cur_event;
      Update the thread's priority basing upon the priority of
      its super event and its original fixed-priority;
     Call Scheduler();
}
```

```
void scheduler (void)
{  /* Determine the scheduling is for events or threads at this
      instant? */
   If (TCB list of cur_event==null or cur_event==daemon)
   then {   /* Event scheduling */
      cur_event = the top one at the ECB ready list;
      cur_thread = the top one at the TCB list of this event;
      Set cur_event / cur_thread to 'execution' state;
      Call cur_thread at cold mode.
   } else {  /* Thread scheduling*/
      If (cur_thread  == the top one at TCB list of this event)
            Return;
      Set cur_thread to 'suspended' state;
      Save_Contect(cur_thread);
      cur_thread = the top one at the TCB list of this event);
      Set Cur_thread to 'execution' state;
      If (the first time calling for current-thread)
            Call cur_thread at cold mode;
      else {  /*wake up from 'suspend' state*/
            Restore_context(cur_thread);
            Call cur_thread at warm mode;
      }
   }
}
```

4.System communicaion & synchronization

The interactions between system components, i.e. event and thread, are essential to provide system communication and synchronization services. According to the action/thread/event structure, LIMOS system provides three system interactions: event-ISR, inter-event and inter-thread.

LIMOS employs *LINDA* concept, i.e. *tuple* space, *IN/OUT* primitive-pair, to facilitate the system interactions. LINDA is a parallel programming language [14]. The basic LINDA primitives of posting and reading *tupl*e are: *IN* & *OUT* [15]. Because a message delivery in LINDA is based on its content matching, the inter-object is thus not space couple. Moreover, since the IPC (inter-process communication) in LINDA is asynchronous, the inter-object is also not time couple. The '*tuple*' model may be extended to support multi-CPU and multi-thread parallel programming model. Therefore LINDA is suitable for parallel distributed applications.

The original LINDA concept is not adequate for distributed hard real-time parallel processing, because the IPC time is not deterministic, which increases dramatically when the number of processes or processors is huge (more than 100) [16]. To overcome these problems, LIMOS provides a simplified tuple space and a light system primitive-pair.

4.1. Tuple space

The tuple space consists of a set of tuples identified by a key. Each tuple contains a critical resource, that is, a special data structure called a circular queue or ring buffer. In the ring buffer, simultaneous input and output of the list are achieved by keeping head and tail indices. Data are loaded at the head and read from the tail. A tuple is encapsulated in a data structure: *Tuple_Table*. The *tuple_ID* is the identifier: key of tuple. A tuple has two states: *ready* and *free*. If the message number, i.e. *tuple_msgnum*, is more than 0, the *tuple_state* is set to 'ready', else is 'free'. The *tuple_size*, *tuple_staadr* and *tuple_endadr* are the size, the start address and the end address of the ring buffer. The *tuple_head* and *tuple_tail* are the writing and reading pointers that have been initialized to the *tuple_staadr* when a tuple is allocated.

```
typedef struct Tuple_Table
{
  char            tuple_ID;       /* tuple identifier: key.*/
  char            tuple_state;    /* tuple state: free 0; ready 1. */
  char            tuple_size;     /* ring buffer size*/
  unsigned char*  tuple_head;     /* writing msg buffer pointer */
  unsigned char*  tuple_tail;     /* reading msg buffer pointer*/
  unsigned long   tuple_staadr;   /* ring buffer start position*/
  unsigned long   tuple_endadr;   /* ring buffer end position*/
  char            tuple_msgnum    /* tuple message number */
```

Instead of matching the entire message content as classic LINDA concept, only one numeric identifier (*KEY*) is used to identify a *tuple* in LIMOS. The numeric identifier and the type of *tuple* are statically assigned to the local, shared or distributed buffer by the user. The tuple ring buffer is mapped into a byte array associated to a *tuple* table. The message content may be accessed immediately when a *tuple* is available. The runtime of *tuple* template matching is thus deterministic in spite of the numbers of events and threads.

In LIMOS, all kinds of data exchanges, no matter interior interactions (between components, i.e. event and thread) or exterior interactions (generally with external peripherals, including sensors, actuator and a variety of interface devices etc), have been implemented via tuple space. Noted that a thread is a component that is a basic system operation unit and an event is the specific kind of component that regards as the container of its sub-threads (multiple components).There is thus no interaction between threads and events.

4.2. IN/OUT operation

The *IN/OUT* primitive-pair in LIMOS contains two main functions: data (*msg*, signal) exchange and transmission via tuple; update the status and priority of event/thread, illuminated in the below pseudo codes.

```
void IN (int  Key,  char * msgPtr, int msgLen)
{
  /*if no data is available, suspend this thread*/
  If (tuple_state !== 'ready')  {/*a new thread scheduling */
    DIS_ALL_IRQ;
    /*up to date the time attributes of threads */
    thread_status = 'suspended';
    thread_deadline = current timeslot + relative
    deadline (lifetime);
    thread_timeslice = relative deadline;
    Save_Context();
    Scheduler();
  }
  /* Step1: data (msg, signal) exchange and transmission */
  Copy data from tuple(key) to received buffer (msgPtr);
  /* Step2: up to date the status of tuple */
  DIS_ALL_IRQ;
  if (--tuple_msgnum ==0)
    tuple_state ='free';
  ENA_ALL_IRQ;
}
```

LIMOS is a *tuple*-based hybrid multi-level system. Each event, no matter activated by a signal from timers, actuators and sensors or from other events, must be associated with a unique *tuple*. When a message received from peripherals or events is arrived, the *OUT* operation is performed to store data into a relative tuple and then to update the status information of the corresponding *tuple*. Consequently, the *OUT* primitive updates the status information of the tuple-relative event / thread and the time slice value of system items in the ECB ready list / TCB suspended list, and then resorts the ECB ready list at sort descending according to their time slice values or sets a thread into the 'terminate' state if the thread runs out of allowable block time.

The reading tuple operation is allowed for events only if data is available on tuple. When an event or thread reads data from its relative tuple on which data is available, the *IN* operation is: to copy data from the tuple to an application, and then update the tuple status. Whereas, if data is not available on tuple, the *IN* operation is: to set the thread into 'suspend' state, and initialize the two time attributes of TCB item, then store the context into the thread stack and activate a new scheduling.

Implementing a system updating within the system primitives may ensure a predictable and determinate system scheduling with no concerning about the number of components, which is the typical feature of hard real-time system. Moreover, when the calling frequency of *IN/OUT* is lower than that of the system scheduling, as in most of WSN applications which has low sampling frequency, the above-mentioned mechanism reduces the system workload efficiently.

```
void OUT(int  Key,  char * msgPtr, int msgLen)
{
   /* Step1: data (msg, signal) exchange and transmission */
   Copy data from sending buffer (msgPtr) to tuple(key);
   /* Step2: update the status & priority of system
essences*/
   DIS_ALL_IRQ;
   this.tuple_msgnum++;
   this.tuple_state ='used';
   if (Key represent an event tuple) {
      this.event_status = 'ready';
      this.event_deadline = current timeslot + relative
      deadline (period);
      this.event_timeslice = relative deadline;
      ENA_ALL_IRQ;
      /*Update timeslice of items of the ECB ready list;*/
      while (ECB ready list is not NULL)
         that.event_timeslice = that.event_deadline –
         current timeslot;
         Add this event into the ECB ready list at the sort
         descending of timeslice;
   } else if (Key represent a thread tuple) {
      this.thread_status = 'ready';
      Add this thread into the TCB ready list at the sort of
      fixed priority;
      ENA_ALL_IRQ;
      /*Update timeslice of items of TCB suspended list;*/
      while (TCB suspended list is not NULL) {
         that.thread_timeslice = that.thread_deadline –
         current timeslot;
         /* timeout: force termination of thread*/
         if (that.thread_timeslice <= 0)
              that.thread_status = 'terminated';
      }
   }
```

5. Evaluation

LIMOS has been ported on ARM7TDMI-S architecture[18] based processors, including NXP LPC21xx[19] and Atmel At91SAM7S series[20] This section estimates the memory and power consumption of LIMOS, calculates the execution time of system primitives, evaluates the system latencies and finally compares the performances of LIMOS with those of other RTOSs amd TinyOS.

5.1. Evaluation Platform

The hardware evaluation platform is an ARM7TDMI-S based 32-bit RISC architecture microcontroller: Atmel At917Sam256. This microcontroller has 256Kbytes internal Flash and 64Kbytes internal SRAM, running at the main operating frequency of 48MHz. It provides two UART controllers, one 8-channel 10-bit ADC, one SSC controller and one SPI interface. The software development platform adopts the IAR system EWARM embedded integrated development environment and J-Link JTAG simulator [21]. The LIMOS instance is configured with three events (two work events and one daemon event). The first work event contains three threads and the second one has two threads.

5.2. Memory and Power Consumption

Since LIMOS is dedicated to strict resource-constrained embedded applications, especially for WSN nodes, it consumes tiny resources, i.e. memory, energy and CPU. Table 1 presents the memory and power consumption of LIMOS on the ARM7.

LIMOS can operate at different operation modes (event-driven, multi-threading), having very little memory requirement (<5Kbytes) comparing with most of RTOSs. Both ARM7 microcontroller and Zigbee RF transceiver support multi-level low-power operation modes. When no component running or no data transmitting, LIMOS can configure associated registers to set MCU or RF transceiver to operate on different low-power mode to reduce power consumption.

Table 1. Memory and power consumption of LIMOS

Memory consumption (bytes)		**Power consumption** (Lithium-Ion battery,3.6V, 1800mAH)		
CODE	**DATA**		**Normal**	**Low-Power**
3572	1272	At91SAM7S (48MHz)	<50mA	< 60µA
		Chipcon CC2340 (Zigbee)	Send mode <25mA	Sleep mode <0.9µA
			Receive mode <27mA	Standby mode <0.6µA
		Lifetime	> 16 hours	> 3 years

For most of WSN applications, such as environmental data collection, security monitoring, and mobile sensor node tracking, the sampling frequency is low and LIMOS is idle at most of the time. Therefore, LIMOS can operate in low-power mode most of the time to reduce the power consumption. In the case of the evaluation platform, the default power source is a Lithium-Ion battery having a capacity of 1800mAH at 3.6V. A WSN system (a real-time continuous data sampling and transmitting application) can run 16 hours in normal mode, whereas in complete low-power mode, the system run-time can be extended to more than 3 years.

5.3. Performance of System primitives

The numbers of instruction cycles of the system primitives, i.e. IN/OUT primitives, on At91SAM7S256 are evaluated. The runtime of each primitive is evaluated by taking into account the running clock frequencies of 48MHz. Table 2 presents the results of performance evaluation of system primitives. In the static configurable LIMOS system, the execution time of system primitives is determined and bounded between the minimal and maximal time. Moreover, the size of a tuple message is limited and predefined, the execution time of IN and OUT primitives are thus predictable. The deterministic and predictable behaviors of system primitives are the key features of a real-time operating system.

Table 2. Performance evaluation of system primitives

Operations	**Cost** (cycles)		**Time** (μs)(48MHz)	
	Max	**Min**	**Max**	**Min**
In	149+46n	95	3.101+ 0.957n	1.977
Operation: Read data from its associated tuple. **Max**: data is available, reading *n* bytes from its associated tuple and copying it into the user buffer. **Min**: no data is available, calling the thread_scheduler function. ***n***: is the byte length of receiving data (n>0).				
Out	104+32n+ Cycle_1	104+32n+ Cycle_2	2.164+ 0.666n+ Cycle_1	2.164+ 0.666n+ Cycle_2
Operation: Send data into its associated tuple, insert an item into the event/thread queue and resort the queue. **Max**: Send data into an event tuple, and insert the related event into the event ready queue. **Min**: Send data into a thread tuple, and insert the related thread into the thread ready queue. ***n***: is the byte length of sending data (n>0). **Cycle_1**: cycles of event enqueue. **Cycle_2**: cycle of thread enqueue.				

5.4. System latencies

For a real-time operating system, the system scheduling scheme and the interrupt handling mechanism are critical, which must be short and deterministic. Several system latencies are used to rate the performances of LIMOS system scheduling and interrupt handling. Since LIMOS adopts action/event/thread structure, therefore there are two latencies that can be used to evaluate system scheduling scheme.

- *event-to-event switch latency*: the time necessary for the system to switch from one event to another. This scheduling operation is performed in the *event_manager* routine. The native thread of next event is called in the 'cold' startup mode, and thus there is no context-switch operation for event-to-event switch.
- *thread-to-thread switch latency*: the time necessary for the system to switch from current thread to another thread of the same event. This scheduling operation is performed in the *thread_scheduler* routine. There is one time context-switch between the system and current thread. If the next thread runs at the first time, it is called in the 'cold' startup mode; whereas, the thread is called in the 'warm' startup mode and there is one more time context switch operation between system and next thread.

Table 3. Performance evaluation of system latencies

Latencies	**Cost** (cycles)			**Time** (μs)(48Mhz)		
	Min	**Avg**	**Max**	**Min**	**Avg**	**Max**
event-to-event switch latency	90			1.873		
	Call *event_manager* routine to run next ready event. The native thread of next event is called in the 'cold' start-up mode					
thread-to-thread switch latency	89[1](99[2])			1.852[1] (2.06[2])		
	Call *thread_scheduler* routine to run next ready thread of current event [1]: Next running thread of this event is called in the 'cold' start-up mode [2]: Next running thread of this event is called in the 'warm' start-up mode					
Interrupt Response Latency	29	107	247	0.604	2.227	5.140
Interrupt Dispatch Latency	32	68	176	0.666	1.415	3.662
	There are three possibilities for the interrupt dispatch operations after the ISR has exited: 1. Return back to the interrupted function. 2. Call *event_manager* to start event scheduling 3. Call *thread_scheduler* to start thread scheduling					

On the other hand, the interrupt latencies are expected to be finite and will never exceed a predefined maximum time. Two latencies are used to rate the performance of the interrupt handling mechanism:

- *Interrupt Response Latency*: the time elapsed between the execution of the last instruction of the interrupted component and the first instruction in the interrupt service routine. This is an indication of the rapidity of the system reaction to an external interrupts.
- *Interrupt Dispatch Latency*: the time interval to go from the last instruction in the interrupt service routine to the next thread scheduled to run. This indicates the time needed to switch from interrupt

mode to user mode

The evaluation results of system switch latencies and interrupt latencies are presented in **Table 3**. The results exposes LIMOS has deterministic and short system switch / interrupt latencies that can satisfy the requirements of most of real-time applications. It's noted that the latencies of system switch are fixed with no concern of the numbers of event/thread.

5.5. Performance comparison

This paper compares LIMOS with other real-time operating systems and TinyOS.

5.5.1. Comparison with other RTOSs. The performance data of some popular RTOSs are obtained from the evaluation reports of Dedicated System Experts. The evaluation reports for the following commercial operating systems are currently available:

- RTX 4.2 from VenturCom, Inc [11].
- Hyperkernel 4.3 from Imagination Systems, Inc[12].
- VxWorks/x86 5.3.1 from WindRiver Systems [4].
- pSOSystem/x86 2.2.6 from Integrated Systems [6].
- QNX 4.25 from QNX Software Ltd[5].
- QNX/Neutrino1.0 from QNX Software Ltd[9].

The evaluation platform adopted by Dedicated Systems Experts is a 200MHz Intel Pentium MMX based PC with a Chaintech motherboard. Time intervals are measured by using external equipment: the PCI bus analyzer; and system peripherals are simulated by means of another external device: PCI bus exerciser. The RTOSs are evaluated with ten different priority-level tasks. The evaluation data of SDREAM is obtained at [2]: SDREAM is evaluated with five periodic tasks, five priority tasks with different priority levels, running on TI TMS320C5410 DSP.

In order to compare the evaluation results with the LIMOS running on At91SAM7S256, we consider that the P200MMX has at least 300MIPS (millions instructions per second) and the At91SAM7S256 has nearly 42MIPS[18]. Hence, the intrinsic execution time of P200MMX is 7 times faster than the At91SAM7S256. The comparisons of two interrupt latencies between other RTOSs and LIMOS are shown in Figure 3. The two histograms show that LIMOS has the smallest average interrupt response latency and the smallest average interrupts dispatch latency.

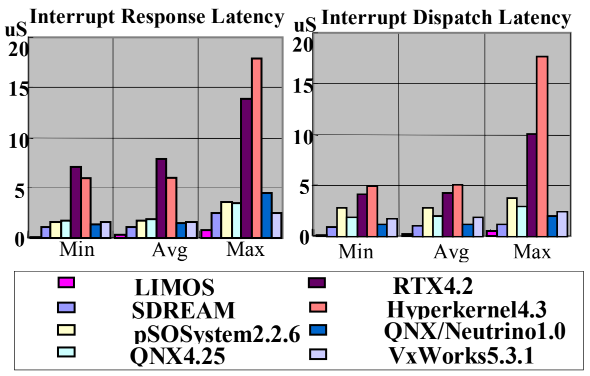


Figure 3. Comparisons of system latencies between LIMOS and other RTOSs

5.5.2. Comparison with other TinyOS. TinyOS is a naturally multitask event-driven system dedicated to wireless sensor applications. It has been tested on ATmega128 (4MHz clock frequency). The evaluation results are currently available 0. Since LIMOS is tested on different platform with TinyOS, this paper only compares three system operations: scheduling a task, context switch and hardware interrupt. The cost of task scheduling indicates the overhead of the thread scheduler function. The operation of context switching happens between the two threads of the same event ('warm mode').

Table 4. Performance evaluation between LIMOS and TinyOS

Operations	**LIMOS(At91SAM7S256)**		**TinyOS(ATmega128)**	
	Cost (cycle)	Time (μs)	Cost (cycle)	Time (μs)
Scheduling task	43	0.895	46	11.5
Context Switch	56	1.165	51	12.75
Hardware Interrupt(hw)	5	0.104	9	2.25
Hardware Interrupt(sw)	61	1.269	71	17.75
OS	CODE size (bytes)		Data size (bytes)	
LIMOS	3572		1272	
TinyOS	432		48	

A hardware interrupt includes the hardware (hw) part and the software (sw) part. Table 4 presents the overheads of three system operations and the memory consumption between LIMOS and TinyOS kernel. Noted that in order to support real-time multitask operations, LIMOS has more system overheads than TinyOS but has similar system cycles for the basic system operations.

6. Conclusion and Perspective

LIMOS is a smart, resource-aware, low-energy and distributed real-time micro-kernel. It adopts the action/event/thread component-based multi-level system architecture. As the result of multi-level structure, LIMOS adopts a two-level scheduling policy: 'non pre-emption priority' high level scheduling for events and 'preemptive priority' low level scheduling for threads. The scheduling scheme is predictable and deterministic with respect to the real-time applications. A unique system interface and a system primitive-pair, i.e. *tuple* and *IN & OUT*, are proposed for system synchronization and communication. LIMOS integrates the advantages of TinyOS and SDREAM. It can be running at different modes: event-driven, multi-tasking. The combination of two kernels greatly extends the application range of LIMOS from simple single-task to multi-task applications.

At present, LIMOS has been ported on the LiveNode hardware platform [22] developed by the University of Blaise Pascal, France. The LivenNode is a specific WSN node that is applied to a variety of WSN applications, including environmental data collection, object tracking and health care, etc.

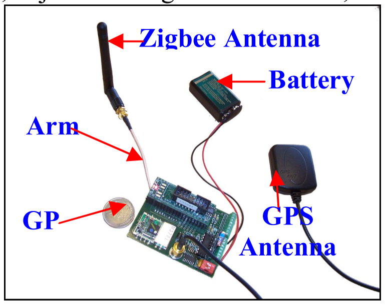


Figure 4. Livenode hardware platform

For the future work, we plan to further improve the performance of LIMOS system at the following aspects:

- Fault-tolerant system: A fault-tolerant system has the ability to continue normal operation despite the presence of hardware or software faults (except for physical destruction). In order to improve the robustness of LIMOS, the ability of fault-tolerant should be taken into consideration carefully.
- Low-energy system: Improving the system energy efficiency is the first essential factor of the WSN system designing. More optimization mechanisms should be adopted to reduce energy consumption.
- Distributed system: A distributed system has the ability of parallel operations on a set of processors or multi-cores of one chip (NoC). The introduction of the tuple concept makes LIMOS suitable for parallel communication and task synchronization on a distributed system.

7. References

[1] W. Maurer. *The Scientist and Engineer's Guide to TinyOS Programming*, Technical Document, http://ttdp.org/tpg/html/, 2004.

[2] H.Y.Zhou, K.M.Hou and C. de Vaulx, *SDREAM: A Super-small Distributed REAl-time Microkernel dedicated to wireless sensors*, Journal of PERVASIVE COMPUT. & COMM. 2006 issue 4 Vol(2), pp398-410.

[3] J.J. Labrosse, *MicroC/OS-II, The Real-time Kernel, R & D Books*, Technical Document; Oct. 1998.

[4] Dedicated Systems Magazine, *VxWorks/x86 5.3.1 evaluation,* http://www.dedicated-systems.com, 2000.

[5] Dedicated Systems Magazine, *QNX4.25 Evaluation Executive Summary*, http://www.dedicated-systems.com, 2000.

[6] Dedicated Systems Magazine, *PSOS 2.2.6 Evaluation Executive Summary*, http://www.dedicated-systems.com, 2000.

[7] LynuxWorks Inc., *Lynxos 4.0: the world's most powerful, open-standards rtos*, http://www.lynuxworks.com/rtos, 2002.

[8] Victor Yodaiken, *An Introduction to RTLinux,* Technical Document, New Mexico Institute of Technology, Oct. 1997.

[9] QNX Software System Ltd., *Qnx neutrion rots microkernel operating system*, Technical Report; MCL-DS-VXW-0208, Feb. 2003.

[10] Microsoft Corporation, *Which to choose: Evaluating Microsoft windows CE.NET and windows XP embedded*, Technical Report, Microsoft Corporation, 2001.

[11] Dedicated Systems Magazine, *RTX4.2 evaluation execute Summary*, http://www.dedicated-systems.com, 2000.

[12] Dedicated Systems Magazine, *Hyperkernel4.3 evaluation execute Summary*, http://www.dedicated-systems.com, 2000.

[13] C.L Liu and J.W.Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, 1973. 20(1): p. 46-61.

[14] A.D.Gelernter, *Generative communication in LINDA. ACM Transactions on Programming Languages and Systems*, 1985. 7(1): p. 80–112.

[15] A.I. Taylor Rowstron, *Bulk Primitives in LINDA run-time systems*, Ph.D thesis; University of York, Oct. 1996.

[16] E. Yao, *Real-time Multiprocess Kernel: Hierarchical LINDA*, Proceedings of ICSPAT. 1993.

[17] P.A.Laplante, *Real-time systems design and analysis : an engineer's handbook*, Wiley & IEEE Press, 3rd ed. 2004.

[18] ARM Ltd., *ARMTDMI-S technical reference manual (v4)*, Technical Document, 2001.

[19] Atmel Ltd., *AT91SAM7S-EK Evaluation Board User Guide*, Technical Document, 2006.

[20] NXP Ltd., *LPC214x User Manual*, Technical Document, 2005.

[21] IAR Ltd., *ARM IAR Embedded Workbench IDE User Guide*. Technical Document, 2006.

[22] K.M.Hou et al. *LiveNode: LIMOS versatile embedded wireless sensor node*, Journal of Harbin Institute of technology, Vol(32), p139-144, Oct. 2007.